Implementing a Notion of Modules in the Logic Programming Language λProlog

Keehang Kwon, Gopalan Nadathur and Debra Sue Wilson

Department of Computer Science, Duke University, NC 27707

Abstract. Issues concerning the implementation of a notion of modules in the higher-order logic programming language λProlog are examined. A program in this language is a composite of type declarations and procedure definitions. The module construct that is considered permits large collections of such declarations and definitions to be decomposed into smaller units. Mechanisms are provided for controlling the interaction of these units and for restricting the visibility of names used within any unit. The typical interaction between modules has both a static and a dynamic nature. The parsing of expressions in a module might require declarations in a module that it interacts with, and this information must be available during compilation. Procedure definitions within a module might utilize procedures presented in other modules and support must be provided for making the appropriate invocation during execution. Our concern here is largely with the dynamic aspects of module interaction. We describe a method for compiling each module into an independent fragment of code. Static interactions prevent the compilation of interacting modules from being completely decoupled. However, using the idea of an interface definition presented here, a fair degree of independence can be achieved even at this level. The dynamic semantics of the module construct involve enhancing existing program contexts with the procedures defined in particular modules. A method is presented for achieving this effect through a linking process applied to the compiled code generated for each module. A direct implementation of the dynamic semantics leads to considerable redundancy in search. We present a way in which this redundancy can be controlled, prove the correctness of our approach and describe run-time structures for incorporating this idea into the overall implementation.

1 Introduction

This paper' concerns the implementation of a notion of modules in the logic programming language λProlog. Logic programming has traditionally lacked devices for structuring the space of names and procedure definitions: within this paradigm, programs are generally viewed as monolithic collections of procedure definitions, with the names of constants and data constructors being implicitly defined and visible everywhere in the program. Although the absence of such facilities is not seriously felt in the development of small programs, structuring mechanisms become essential for programming-in-the-large. This fact has spurred investigations into mechanisms for constructing programs in a modular fashion (*e.g.*, see [11, 14, 19, 20]) and has also resulted in structuring devices being included in some implementations of a Prolog-like language on an *ad hoc* basis. Most proposals put forth have, at the lowest level, been based on the use of the logic of Horn clauses. This logic does not directly support the realization of structuring devices, and consequently these have had to be built in at an extra-logical level. The logic of hereditary Harrop formulas, a recently discovered extension to Horn clause logic [13], is interesting in this respect because it contains logical primitives for controlling the visibility of names and the availability of predicate definitions. The language λProlog is based on this extended logic and thus provides logical support for several interesting scoping constructs [10, 11]. The notion of modules whose implementation we describe in this paper is in fact based on these new mechanisms.

The language λProlog is in reality a typed language. One manifestation of this fact is that programs in this language consist of two components: a set of type declarations and a set of procedure definitions. The module concept that we consider is relevant to a structuring of programs with respect to both components. In a simplistic sense, a module corresponds to a named collection of type declarations and procedure definitions. This view of modules reveals that the use of this structuring notion has both static and dynamic effects. The typical use that might be expected of any module is that of making it contents available in some fashion within a program context such as another module. The main impact of making the declarations in a module visible must clearly be a static one: to take one example, the type associated with some constant by the module in question may be needed for parsing expressions in the new context. The effect with regard to predicate definitions is, on the other hand, largely dynamic. Thus, procedure definitions in the new context might contain invocations to procedures defined in the "imported" module. The important question to be resolved, then, is that of how a reference to code is to be resolved in a situation where the available code is changing dynamically.

From the perspective of implementing the module notion, the main concern is really with the dynamic aspects. In particular, our interest is largely in a method for compiling the definitions appearing in modules and in the run-time structures needed for implementing the prescribed semantics for this construct. We examine these questions in detail in this paper and suggest solutions to them. Now, λProlog has several new features in comparison with a language such as Prolog and a complete treatment of compilation requires methods to be presented for handling these features as well. We have studied the implementation issues arising out the other extensions in recent work and have detailed solutions to them [7, 16, 17]. We outline the nature of these solutions here but do not present them in detail. In a broad sense, our solutions to the other problems can be embedded in a machine like the Warren Abstract Machine (WAM) [21]. We start with this machine and describe further enhancements to it that serve to implement the dynamic aspects of the module notion. There are several interesting characteristics to the scheme we ultimately suggest for this purpose, and these include the following:

(i) A notion of separate compilation for modules is supported. As we explained above, there is a potential for static interaction between modules that makes completely independent compilation impossible. However, this situation is no different from that in any other programming language. We propose the idea of an interface definition to overcome this problem. Relative to such definitions, we show that the separate compilation goal can actually be achieved.

(ii) A notion of linking is described and implemented. The dynamic use of modules effectively reduces to solving goals of the form M ==> G where M is a module name. The expected action is to enhance an existing program context with the definitions in M before solving G. The symbol ==> can, in a certain sense be viewed as a primitive for linking the compiled code generated for a module into a program context. Using ideas from [8] and [16] we show how this primitive can be implemented.

(iii) A method for controlling redundancy in search is described. The dynamic semantics presented for modules in [11] can lead to the definitions in a module being added several times to a program context, leading to considerable redundancy in solving goals. We present a sense in which this redundancy can be eliminated, prove the correctness of our approach and show how this idea can be incorporated into the overall implementation. The general idea in avoiding redundancy has been used in earlier implementations of λProlog [2, 9]. However, ours is, to our knowledge, the first proof of its correctness and the embedding of the idea within our compilation model is interesting in its own right.

The remainder of this paper is structured as follows. We describe the language of λProlog without the module feature in the next section, focussing eventually on the general structure of an implementation for this "core". In Section 3, we present the module notion that is the subject of this paper and outline the main issues in its implementation. In Section 4, we present our first implementation scheme. This scheme permits separate compilation and contains the run-time devices needed for linking. However, it has the drawback that it is may add several copies of a module to a program context leading to the mentioned redundancy in search. We discuss this issue in detail in Section 5 and show a way in which redundancy can be controlled. In Section 6 we use this idea in describing mechanisms that can be incorporated into the basic scheme of Section 4 to ensure that only one copy of a module is available in a program context at any time. Section 7 concludes the paper.

2 The Core Language

We describe in this section the part of the λProlog language that can be thought of as its core. Our presentation will be at two levels: we shall describe the logical underpinnings of the language and also attempt to describe it at the level of a usable programming language. Both aspects are required in later sections. The exposition at a logical level are needed to understand the semantics of the modules notion and to justify optimizations in its implementation. The presentation of the programming language is necessary to understand the value of modules as a pragmatic structuring construct.

1.1 Syntax

The logical language that underlies λProlog is ultimately derived from Church's simple theory of types [1]. This language is *typed* in the sense that every well-formed expression in it has a type associated with it. The language of types that is actually used permits a form of polymorphism. The type expressions are obtained from a set of *sorts*, a set of *type variables* and a set of *type constructors*, each of which is specified with a unique arity. The rules for constructing types are the following: (i) each sort and type variable is a type, (ii) if c is an n-ary type constructor and $t_1, \ldots, t_n$ are types, then $(c\ t_1\ \ldots\ t_n)$ is a type, and (iii) if α and β are types then so is $\alpha \rightarrow \beta$. Types formed by using (iii) are called *function* types. In writing function types, parentheses can be omitted by assuming that $\rightarrow$ is right associative. Type variables have a largely abbreviatory status in the language: they can appear in the types associated with expressions, but at a conceptual level such expressions can be used in a computation only after all the type variables appearing in them have been instantiated by closed types. A type is closed if it contains no type variables. However, these variables permit a succinct presentation of predicate definitions and, as we mention later, their instantiations at run-time can often be delayed. Thus, type variables provide a sense of polymorphism in λProlog.

At the level of concrete syntax, type variables are denoted by names that begin with an uppercase letter. The set of sorts initially contains only o, the boolean type, and *int*, the type of integers, and no type constructors are assumed. The user can define type constructors by using declarations of the form

kind c $\quad$ *type* $\rightarrow \ldots \rightarrow$ *type.*

The arity of the constructor c that is thus declared is one less than the number of occurrences of *type* in the declaration. Noting that a sort might be viewed as a nullary type constructor, a declaration of the above kind may also be used to add new sorts. As specific examples, the declarations

kind *i* $\quad$ *type.*
kind *list* $\quad$ *type* $\rightarrow$ *type.*

add *i* to the set of sorts and define *list* as a unary constructor. The latter will be used below as a means for constructing types corresponding to lists of objects of a homogeneous type .

The *terms* of the language are constructed from given sets of constant and variable symbols, each of which is assumed to be specified with a type. The constants are categorized as the *logical* and the *nonlogical* ones. The logical constants consist of the following:

true	of type o, denoting the true proposition,
$\wedge$	of type $o \rightarrow o \rightarrow o$, representing conjunction,
$\vee$	of type $o \rightarrow o \rightarrow o$, representing disjunction,
$\supset$	of type $o \rightarrow o \rightarrow o$, representing implication,
sigma	of type $(A \rightarrow o) \rightarrow o$, representing existential quantification,
pi	of type $(A \rightarrow o) \rightarrow o$, representing universal quantification.

The symbols *sigma* and *pi* have a polymorphic type associated with them. These symbols really correspond to a family of constants, each indexed by a choice of ground instantiation for τ and a similar interpretation is intended for other polymorphic symbols.

In the machine presentation of nonlogical constants and variables, conventions similar to those in Prolog are used: both variables and constants are represented by tokens formed out of sequences of alphanumeric characters or sequences of "sign" characters, and those tokens that begin with uppercase letters correspond to variables. The underlying logic requires a type to be associated with each of these tokens. Symbols that consist solely of numeric characters are assumed to have the type *int*. For other symbols, an association is achieved by declarations of the form

type *constant* *type-expression*.

Such a declaration identifies the type of *constant* with the corresponding type expression. As examples, the declarations

type *nil* $(list\ A)$.
type :: $A \rightarrow (list\ A) \rightarrow (list\ A)$.

define the constants *nil* and :: that function as constructors for homogeneous lists. Types of constants and variables may also be indicated by writing them in juxtaposition and separated by a colon. Thus the notation X : *int* corresponds to a variable X of type *int*.

The terms in our logical language are obtained from the constant and variable symbols by using the mechanisms of function abstraction and application. In particular (i) each constant and variable of type τ is a term of type τ, (ii) if x is a variable of type τ and t is a term of type τ', then $\lambda x t$ is a term of type $\tau \rightarrow \tau'$, and (iii) if t_1 is a term of type $(\tau_2 \rightarrow \tau_1)$ and t_2 is a term of type τ_2, then $(t_1\ t_2)$ is a term of type τ_1. A term obtained by virtue of (ii) is referred to as an *abstraction* whose bound variable is x and whose scope is t. Similarly a term obtained by (iii) is called the application of t_1 to t_2.

Several conventions are adopted towards enhancing readability. Parentheses are often omitted by assuming that application is left associative and that abstraction is right associative. The logical constants $\wedge$, $\vee$ and $\supset$ are written as right associative infix operators. It is often useful to extend this treatment to nonlogical constants, and a device is included in λProlog for declaring specific constants to be prefix, infix or postfix operators. For instance, the declaration

infix 150 *xfy* :: .

achieves the same effect that the declaration $op(150, xfy, ::)$ achieves in Prolog: it defines :: to be a right associative infix operator of precedence 150.

An important notion is that of a ***positive*** term which is a term in which the symbol $\supset$ does not appear. We define an *atomic formula* or *atom* to be a term of type o that has the structure $(P\ t_1\ \ldots\ t_n)$ where P, the *head* of the atom, is either a nonlogical constant or a variable and $t_1, \ldots, t_n$, the *arguments* of the atom, are positive terms. Such a formula is referred to as a *rigid* atom if its head is a nonlogical constant, and as a *flexible* atom otherwise. Using the symbol A to denote arbitrary atoms and A_r to denote rigid atoms, the classes of G-, D- and E-formulas are identified as follows:

$$\begin{aligned} G &::= \mathit{true} \mid A \mid (G_1 \wedge G_2) \mid (G_1 \vee G_2) \mid \mathit{sigma}\ (\lambda x G) \mid \\ &\qquad \mathit{pi}\ (\lambda x G) \mid (E \supset G) \\ D &::= A_r \mid G \supset A_r \mid \mathit{pi}\ (\lambda x D) \mid (D_1 \wedge D_2) \\ E &::= D \mid \mathit{sigma}\ (\lambda x E) \end{aligned}$$

A curious aspect of these syntax rules is the use of the symbols *pi* and *sigma*. These symbols represent universal and existential quantification respectively. The quantifiers that are used in conventional presentations of logic play a dual role: in the expression $\forall x P$, the quantifier has the function of binding the variable x over the expression P in addition to that of making a predication of the result. In the logical language considered here, these roles are separated between the abstraction operation and appropriately chosen constants. Thus the expression $\forall x P$ is represented here by $(pi\ (\lambda x P))$. The former expression may be thought of as an abbreviation for the latter, and we use this convention at a metalinguistic level below. A similar observation applies to the symbol *sigma* and existential quantification.

The G- and D-formulas determine the *programs* and *queries* of λProlog. A program consists of a list of closed D-formulas each element of which is referred to as a *program clause*, and a query or goal is an closed G-formula[1]. In writing the program clauses in a program in λProlog, the universal quantifiers appearing at the front are left implicit. A similar observation applies to the existential quantifiers at the beginning of a query. There are some other conventions used in the machine presentation of programs. Abstraction is depicted by \, written as an infix operator. Thus, the expression $\lambda X(X :: nil)$ is represented by $X \backslash (X :: nil)$. The symbols $\wedge$ and $\vee$ are denoted by , and ; as in Prolog. Implications appearing at the top-level in program clauses are written backwards with `:-` being used in place of $\supset$, and the symbol $\supset$ in goal formulas is written as `=>`. Finally, a program is depicted by writing a sequence of program clauses, each clause being terminated by a period. An example of the use of these conventions is provided by the following clauses defining the familiar *append* predicate, assuming the types for *nil* and :: that were presented earlier.

[1] This definition is more general than the one usually employed in that existential quantification is permitted over D formulas appearing to the left of implications in goals. This feature does not add anything new at a logical level, but is pragmatically useful as we see later. This extended definition is also used in [4].

(*append nil L L*).
(*append H :: L*1 *L*2 *H :: L*3) :- (*append L*1 *L*2 *L*3).

Notice that not all the needed type information has been presented in these clauses: the types of the variables and of *append* have been omitted. These types could be provided by using the devices explained earlier. However, type declarations can be avoided in several situations since the desired types can be reconstructed [15]. For example, the type of *append* in the above program can be determined to be $(list\ A) \rightarrow (list\ A) \rightarrow (list\ A) \rightarrow o$. The type reconstruction algorithm that is used is sensitive to the set of clauses contained in the program. For example, if the program above included the clause

(*append* (1 :: *nil*) (2 :: *nil*) (1 :: 2 :: *nil*)).

as well, then the type determined for *append* would be $(list\ int) \rightarrow (list\ int) \rightarrow (list\ int) \rightarrow o$ instead.

The example above shows the similarity of λProlog syntax to that of Prolog. The main difference is a curried notation, which is convenient given the higher-order nature of the language. There are similarities in the semantics as well as we discuss below.

1.1 Answering Queries from Programs

We present an operational semantics for λProlog by providing rules for solving a query in the context of a given program. The rules depend on the top-level logical symbol in the query and have the effect of producing a new query and a new program. Thus, the operational semantics induces a notion of computational state given by a program and a query. We employ structures of the form $\mathcal{P} \longrightarrow G$ where $\mathcal{P}$ is a listing of closed program clauses and G is a closed G-formula to represent such a state. We refer to these structures as ***sequents***, and the idea of solving a query from a set of closed program clauses corresponds to that of constructing a derivation for an appropriate sequent.

Several auxiliary notions are needed in presenting the rules for constructing derivations. One of these is the notion of equality assumed in our language. Two terms are considered equal if they can be made identical using the rules of λ-conversion. We assume a familiarity on the part of the reader with a presentation of these rules such as that found in [5]. We need a substitution operation on formulas. Formally, we think of a substitution as a finite set of pairs of the form $\langle x, t\rangle$ where x is a variable and t is a term whose type is identical to that of x; the substitution is said to be closed if the second component of each pair in it is closed. Given a substitution $\{\langle x_i, t_i\rangle | 1 \leq i \leq n\}$, we write $F[t_1/x_1, \ldots, t_n/x_n]$ to denote the application of this substitution to F. Such an application must be done carefully to avoid the usual capture problems. The needed qualifications can be captured succinctly by using the λ-conversion rules: $F[t_1/x_1, \ldots, t_n/x_n]$ is equal to the term $((\lambda x_1 \ldots \lambda x_n F)\ t_1\ \ldots\ t_n)$. We also need to talk about ***type instances*** of terms. These are obtained by making substitutions for type variables that appear in the term. Finally, we are particularly interested in terms that do not have any type variables in them and we call such terms *type variable free.*

The various notions described above are used in defining the idea of an instance of a program clause.

Definition 1. An instance of a closed program clause D is given as follows:

(i) If D is of the form A_r or $G \supset A_r$, then any type variable free type instance of D is an instance of D.
(ii) If D is of the form $D_1 \wedge D_2$ then an instance of D_1 or of D_2 is an instance of D.
(iii) If D is of the form $\forall x D_1$, then an instance of $D_1[t/x]$ for any closed positive term t of the same type as x is an instance of D.

The restriction to (closed) positive terms forces an instance of a program clause to itself be a program clause. In fact, instances of program clauses have a very simple structure: they are all of the form A_r or $G \supset A_r$.

In describing the derivation rules, and thus the operational semantics of our language, we restrict our attention to type variable free queries. We present a more general notion of computation later based on this restricted definition of derivation.

Definition 2. Let G be a type variable free query and let $\mathcal{P}$ be a program. Then a derivation is constructed for $\mathcal{P} \longrightarrow G$ by using one of the following rules:

SUCCESS	By noting the G is equal to an instance of a program clause in $\mathcal{P}$.
BACKCHAIN	By picking an instance of a program clause in $\mathcal{P}$ of the form $G_1 \supset G$ and constructing a derivation for $\mathcal{P} \longrightarrow G_1$.
AND	If G is equal to $G_1 \wedge G_2$, by constructing derivations for the sequents $\mathcal{P} \longrightarrow G_1$ and $\mathcal{P} \longrightarrow G_2$.
OR	If G is equal to $G_1 \vee G_2$, by constructing a derivation for either $\mathcal{P} \longrightarrow G_1$ or $\mathcal{P} \longrightarrow G_2$.
INSTANCE	If G is equal to $\exists x G_1$, by constructing a derivation for the sequent $\mathcal{P} \longrightarrow G_1[t/x]$, where t is a closed positive term of the same type as x.
GENERIC	If G is equal to $\forall x G_1$, by constructing a derivation for the sequent $\mathcal{P} \longrightarrow G_1[c/x]$, where c is a nonlogical constant of the same type as x that does not appear in $\forall x G$ or in the formulas in $\mathcal{P}$.
AUGMENT	If G is equal to $(\exists x_1 \ldots \exists x_n D) \supset G$, by constructing a derivation for $D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P} \longrightarrow G$, where, for $1 \leq i \leq n$, c_i is a nonlogical constant of the same type as x_i that does not appear in $(\exists x_1 \ldots \exists x_n D) \supset G$ or in the formulas in $\mathcal{P}$.

To understand the operational semantics induced by these rules, let us assume a program given by the following clauses

```
(rev L1 L2) :-
        (((rev_aux nil L2),
        (pi (X\(pi (L1\(pi (L2\
                ((rev_aux X :: L1 L2) :- (rev_aux L1 X :: L2)))))))))
                => (rev_aux L1 nil)).
```

and consider solving the query (*rev* 1 :: 2 :: *nil* 2 :: 1 :: *nil*). The first rule that must be used in a derivation is BACKCHAIN. Using it reduces the problem to that of solving the query

```
    ((rev_aux nil 2 :: 1 :: nil),
    (pi (X\(pi (L1\(pi (L2\
            ((rev_aux X :: L1 L2) :- (rev_aux L1 X :: L2)))))))))
    => (rev_aux 1 :: 2 :: nil nil)
```

from the same program. The AUGMENT rule is now applicable and using it essentially causes the program to be enhanced with the clauses

```
    (rev_aux nil 2 :: 1 :: nil).
    (rev_aux X :: L1 L2) :- (rev_aux L1 X :: L2).
```

prior to solving the query (*rev_aux* 1 :: 2 :: *nil nil*). Using the BACKCHAIN rule twice in conjunction with the last clause produces the goal (*rev_aux nil* 2 :: 1 :: *nil*). The derivation attempt now succeeds because the goal is an instance of program clause.

The above example indicates the programming interpretation given to logical formulas and symbols by the operational semantics. Program clauses of the form $\forall x_1 \ldots \forall x_n A_r$ and $\forall x_1 \ldots \forall x_n (G \supset A_r)$ function in a sense as procedure definitions: the head of A_r represents the name of the procedure and, in the latter case, the body of the clause, G, corresponds to the body of the procedure. From an operational perspective, every program clause is equivalent to a conjunction of clauses in this special form, and a program is equivalent to a list of such clauses. Thus both correspond to a collection of procedure definitions. Goals correspond to search requests with the logical symbols appearing in them functioning as primitives for specifying the search structure. Thus, in searching for a derivation, $\wedge$ gives rise to an AND branch, $\vee$ to an OR branch and *sigma* to an OR branch parameterized by a substitution. These symbols are used in a similar fashion in Prolog. The symbols $\supset$ and *pi*, on the other hand, do not appear in Prolog goals. The treatment of these symbols is interesting from a programming viewpoint. The first symbol has the effect of augmenting an existing program for a limited part of the computation. Thus, this symbol corresponds to a primitive for giving program clauses a scope. The symbol *pi* similarly corresponds to a primitive for giving names a scope; processing this symbol requires a new name to be introduced for a portion of the search. A closer look at the operational semantics reveals a similarity between the interpretation of *pi* and the treatment given to existential quantifiers in E-formulas through the AUGMENT rule. This is not very surprising since the formulas $\forall x(D(x) \supset G)$ and $(\exists x D(x)) \supset G$ are equivalent in most logical contexts, assuming x does not appear free in G. From a pragmatic perspective, then, the existential quantifier in E-formulas enables a name to be made local to a set of procedure definitions, *i.e.*, it provides a means for information hiding.

A computation in λProlog corresponds to constructing a derivation for a query from a given program. We are generally interested in extracting a value from a computation. In the present context, this can be made clear as follows.

Definition 3. Let $\mathcal{P}$ be a collection of program clauses and let G be a type variable free query of the form $\exists x_1 \ldots \exists x_n G_1$; the variables $x_1, \ldots, x_n$ are assumed to be implicitly quantified here. An answer to G in the context of $\mathcal{P}$ is a closed substitution $\{\langle x_i, t_i \rangle | 1 \leq i \leq n\}$ such that $\mathcal{P} \longrightarrow G_1[t_1/x_1, \ldots, t_n/x_n]$ has a derivation.

In general our queries may have type variables in them. The answers to such a query are given by the answers to each of its type variable free type instances.

Our ultimate interest is in a procedure for carrying out computations of the kind described above and for extracting results from these. The rules for constructing derivations provide a structure for such a procedure but additional mechanisms are needed. One problem involves instantiations for type variables. There is usually insufficient information for choosing instantiations for these at the points indicated. This problem can be overcome by allowing type variables into the computation and by using unification to incrementally determine their instantiations. A similar problem arises with existential quantifiers in queries. For example, solving a query of the form $\exists x G$ requires a closed term t to be produced that makes $G[t/x]$ solvable. The usual mechanism employed in these cases is to replace x with a *logic* variable, *i.e.*, a place-holder, and to let an appropriate instantiation be determined by unification. However, this mechanism must be used with care in the present situation. First, the unification procedure that is used must incorporate our enriched notion of equality, *i.e.*, higher-order unification [6] must be used. Second, the treatment of universal quantifiers requires unification to respect certain constraints. For example, consider the query $\exists x \forall y p(x, y)$, where p is a predicate constant. Using the mechanisms outlined, this query will be transformed into $p(X, c)$, where c is a new constant and X is a logic variable. Notice, however, that X must not be instantiated with a term that contains c in it. A solution to this problem is to add a numeric tag to every constant and variable and to use these tags in constraining the unification process [3, 18].

A suitable abstract interpreter can be developed for λProlog based on the above ideas[2]. In actually implementing this interpreter, two additional questions arise. First, there is some nondeterminism involved: in solving an atomic goal, a choice has to be made between program clauses and in solving $G_1 \vee G_2$ a decision has to be made between solving G_1 and G_2. The usual device employed here is to use a depth-first search with backtracking. The second question concerns the implementation of implications in queries. To understand the various problems

[2] Actually, the proper treatment of type variables in a computation is still an open issue. However, a discussion of this matter is orthogonal to our present purposes.

that arise here, let us consider a query of the form $(D \supset G_1) \wedge G_2$. This query results in the query $D \supset G_1$ which must be solved by adding (the clauses in) D to the program, solving the clauses in G_1 and then removing D. The addition of code follows a stack based discipline and can be implemented as such. However, if a compilation model is used, some effort is involved in spelling out a scheme for achieving the addition and deletion of code. Moreover the "program clauses" that are added might now contain logic variables in them. Thus, consider solving the goal $\exists L(rev\ 1 :: 2 :: nil\ L)$ using the clause for *rev* presented earlier in this section. The program would at a certain stage have to be augmented with the clause $(rev\_aux\ nil\ L)$ where L is now a logic variable. In general, we need now to think of procedures as blocks of code *and* bindings for some variables. Continuing now with the solution of the query $(D \supset G_1) \wedge G_2$, the goal G_2 will be attempted after the first conjunct is solved. A failure in solving this goal might require an alternative solution to G_1 to be generated. Notice, however, that an attempt to find such a solution must be made in a context where the program once again contains D. An implementation of our language must support the needed context switching ability.

Implementation techniques have been devised for solving the various problems mentioned above [7, 16, 17], resulting in an abstract machine and a compilation scheme for the core language described in this section. We do not discuss this explicitly here, and will rely on the reader's intuition and indulgence when alluding to these ideas later in the paper. However, the discussion of modules will require a closer acquaintance with the scheme used for implementing implications in queries, and we then supply some further details.

Before concluding this section, it is interesting to note the connection between our notion of computation and deduction in a logical context. The following proposition describes this connection.

Proposition 4. *Let $\mathcal{P}$ be a program and let $\mathcal{P}'$ be the collection of all the type variable free type instances of formulas in $\mathcal{P}$. Further, let G be a type variable free query. Then there is a derivation for $\mathcal{P} \longrightarrow G$ if and only if G follows from $\mathcal{P}'$ in intuitionistic logic.*

Only the *only if* part of this proposition is non-trivial. For the most part, this follows from the existence of uniform proofs for sequents of the kind we are interested in; see, *e.g.*, [13] and [18] for details. One additional point to note is the treatment of existential quantifiers in E-formulas. However, this causes no problem because the introduction of existential quantifiers in assumptions can always be made the last step in intuitionistic proofs.

3 Modules

The language described thus far only permits programs that are a monolithic collection of kind, type, and operator declarations together with a set of procedure definitions. Modules provide a means for structuring the space of declarations and also for tailoring the definitions of procedures depending on the context.

The ultimate purpose of this feature is to allow programs to be built up from logical segments which are in some sense separate.

At the very lowest level, the module feature allows a name to be associated with a collection of declarations and program clauses. An example of the use of this construct is provided by the following sequence of declarations that in effect attaches the name *lists* with the list constructors and some basic list-handling predicates:

```
module      lists.
infix 150   ::      xfy.
kind  list  type → type.
type  nil   (list A).
type  ::    A → (list A) → (list A).

(append nil L L).
(append (H :: L1) L2 (H :: L3)) :- (append L1 L2 L3).

(member H (H :: L)).
(member X (H :: L)) :- (member X L).

(length 0 nil).
(length N (H :: L) :- ((length N1 L), N is N1 + 1).
```

One way to think of this module declaration is as a declaration of a list "data type". This data type can be made available in specific contexts by using the name *lists* in a manner that we describe presently. This discussion will bring out the intended purpose of the modules feature. However, there is one use that can already be noted. Looking at the *lists* module above, we see that the types of the predicates defined in it have not been provided. These types can be reconstructed, but, as we noted in Section 2, the types "inferred" depend on the set of available program clauses. The module boundary provides a notion of scope that is relevant to this reconstruction process: looked at differently, the types of all the symbols appearing in the clauses in a module are completely determined once the module is parsed.

The meaning of the module feature is brought out by considering its use in programming. In the presence of modules, we enhance our goals to include a new kind of expression called a ***module implication***. These are expressions of the form M ==> G, where M is a module name. Goals of the new sort have the intuitive effect of adding M to the program before solving G. In making this precise, however, the effect of M on two different components have to be made clear: on the type, kind and operator declarations and on the procedure definitions.

The effect on the space of declarations that we assume here is simple. All the associations present in M become available on adding M to the context. This is really a *static* effect in that it provides a context in which to parse the goal G in a larger goal M ==> G. As a concrete example, consider the goal

```
lists ==> (append 1 :: 2 :: nil  3 :: nil L).
```

In parsing this query, there is a need to determine the types of *append* and of ::. The semantics attributed to the modules feature requires the types associated with these tokens in the module *lists* to be assumed for this purpose. This appears to be the most natural course, given that we expect the definition of *append* provided in *lists* to be useful in solving this query.

From the perspective of procedure definitions, we assume the semantics for modules that is presented in [11]. Within this framework, the dynamic aspects of the module feature are explained by a translation into the core language. Thus, a module is thought of as the conjunction of the program clauses appearing in it. For instance, the *lists* module corresponds to the conjunction of the clauses for *append*, *member* and *length*. Under this interpretation, a module corresponds to a D-formula as described in the last section. Now if module M corresponds to the formula D, the query M ==> G is thought of as the goal D => G. The run-time treatment of module implication is then determined by the AUGMENT rule presented in the last section. In particular, solving the goal M ==> G calls for solving the goal G after adding the predicate definitions in the module M to the existing program.

The analogy between a module and a data type raises the question of whether some aspects of an implementation might be hidden. Our language permits constant names to be made local to a module, thus allowing for the hiding of a data structure. To achieve this effect, a declaration of the form

local *constant*, ..., *constant*.

can be placed within a module. The names of the constants listed then become unavailable outside the module. For example, adding the declaration

local ::.

to the *lists* module has the effect of hiding the list constructor ::.

The static effect of the local construct is easy to understand: only some names are available when the module is added to a context. From a dynamic perspective, another issue arises. Can constants defined to be local eventually become visible outside through computed answers? The expectation is that they should not become so visible. This effect can be achieved by thinking of local constants really as variables quantified existentially over the scope of the conjunction of program clauses in the module. As an example, consider the following module

```
module      store.
local   emp, stk.
kind    store   type → type.
type    emp     (store A).
type    stk     A → (store A) → (store A).
initialize emp.
(enter X S (stk X S)).
(remove X (stk X S) S).
```

This module implements a *store* data type with initializing, adding and removing operations. At a level of detail, the store is implemented as a stack. However, the intention of the local declarations is to hide the actual representation of the store. Now, from the perspective of dynamic effects, the module corresponds to the formula

```
∃Emp∃Stk(
        (initialize Emp) ,
        (pi (X\(pi (S\(enter X S (Stk X S)))))),
        (pi (X\(pi (S\(remove X (Stk X S) S)))))).
```

This formula has the structure of an E-formula and in fact every module corresponds in the sense explained to an E-formula. Referring to this formula as *EStore*, let us consider solving a goal of the form $\exists X(\mathit{Store} \Longrightarrow G(X))$. The semantics of this goal requires solving the goal $\exists X(\mathit{EStore} \Rightarrow G(X))$. Under the usual treatment of existential quantifiers, this results in the goal $(\mathit{EStore} \Rightarrow G(X))$ where X is now a logic variable. Using the AUGMENT rule, this goal is solved by instantiating the existential quantifiers at the front of *EStore*, adding the resulting D-formula to the program and then solving $G(X)$. The important point to note now is that any substitution that is considered for X must not have the constants supplied for *Emp* and *Stk* appearing in it. Thus the semantics attributed to modules and local declarations achieves the intended dynamic effect.

While module implication is useful for making modules available at the top-level, modules may themselves need to interact. For instance, a module that contains sorting predicates might need the declarations and procedure definitions in the *lists* module and a module that implements graph-search might similarly need the *store* and *lists* modules. The needed interaction is obtained in λProlog by placing an *import* declaration in the module which needs other modules. The format of such a declaration is the following:

import $M1, \ldots, Mk$.

In a declaration of this sort, $M1, \ldots, Mk$ must be names of other modules that are referred to as the *imported* modules. A declaration of this sort has, once again, a static and a dynamic effect on the module in which it is placed, *i.e.*, the *importing* module. The static effect is to make all the declarations in the imported modules, save those hidden by local declarations, available in the importing module. These declarations can be used in parsing the importing module and also become part of the declarations provided by that module. The intended dynamic effect, on the other hand, is to make the procedure definitions in the imported modules available for solving the goals in the bodies of program clauses that appear in the importing module. This effect can actually be explained by using module implication [11]. Let us assume that the clause P :- G appears in a module that imports the modules $M1, \ldots, Mk$. The dynamic semantics involves interpreting this clause as the following one instead:

P :- $(M1 \Longrightarrow \cdots (Mk \Longrightarrow G))$.

Observe that using this clause involves solving the goal ($M1$ ==> $\cdots$ (Mk ==> G)) that ultimately causes the program to be enhanced with the clauses in $M1, \ldots, Mk$ before solving G.

```
module graph_search.
import lists, store.

(g_search Soln) :-
          ((init_open Open), (expand_graph Open nil Soln)).

(init_open Open) :-
          ((start_state State), (initialize Op), (enter State Op Open)).

(expand_graph Open Closed Soln) :-
          (remove State Open ROp),
          (((final_state State), (soln State Soln));
          ((expand_node State NStates),
           (add_states NStates ROp (State :: Closed) NOp),
           (expand_graph NOp (State :: Closed) Soln))).

(add_states nil Open Closed Open).
(add_states (St :: RSts) Open Closed NOpen) :-
          ((member St Closed), (add_states RSts Open Closed NOpen)).
(add_states (St :: RSts) Open Closed NOpen) :-
          ((enter St Open NOp), (add_states RSts NOp Closed NOpen)).

          ...
```

Fig. 1. A Module Implementing Graph Search

The definition of the module *graph_search* presented in Figure 1 illustrates the usefulness of the module interaction facility provided by *import*. The definitions of the predicates *start_state*, *final_state*, *soln* and *expand_node* have not been presented here, but we anticipate the reader can supply these. The important aspect to note is the use that is made of the declarations and procedure definitions in the modules *lists* and *store*. For example, the type

$$(list\ A) \rightarrow (store\ A) \rightarrow (list\ A) \rightarrow (store\ A) \rightarrow o$$

will be reconstructed for *add_states*. This type uses type constructors defined in in the modules *lists* and *store*. Similarly, the procedure *member* defined in *lists* and the procedures *initialize*, *enter* and *remove* defined in *store* are used in the program clauses in the module *graph_search*. A particularly interesting aspect is the interaction between the modules *graph_search* and *store*. Notice that the "constants" *emp* and *stk* used in *store* are not visible in *graph_search* and cannot be used explicitly in the procedures appearing there. Thus, importing *store* provides an abstract notion of a store without opening up the actual implementation. For example, the current stack-based realization of the store can be replaced by a queue-based one without any need to change the *graph_search* module so long as the operations *initialize*, *enter* and *remove* are still supported. This change will have an effect on the behavior of *g_search* though, changing it to a procedure that conducts breadth-first search as opposed to the current depth-first search.

The pragmatic utility of the module feature and of the scoping ability provided by the new logical symbols in our language is an important issue to consider and detailed discussions of this aspect appear in [10] and [11]. Our interest in this paper is largely on implementation issues, especially those arising out of the module notion. From this perspective, it is necessary to understand carefully the dynamic interactions that can arise between modules through the use of the *import* statement. We therefore present an example that illustrates some of these interactions. Figure 2 contains a collection of interacting modules and Figure 3 exhibits the process of solving the query ($m1$ ==> (p X)) given these definitions. In presenting this solution attempt, we use a linear format based on the notion of derivation presented in Section 2 but augmented with the use of logic variables. Further, we use lines of the following form

$M1, \ldots, Mn$?- $G(X)$

where $G(X)$ is an atomic goal and $M1, \ldots, Mn$ are module names. Such a line indicates that $G(X)$ is to be solved from a program given by the collection of clauses in $M1, \ldots, Mn$. We refer to this list of modules as a program context. Now, the attempt to solve this goal proceeds by trying to match the goal with the head of some clause. If this attempt is successful, the line is annotated by a binding for the logic variables, *e.g.*, by an expression such as X <- a. In the case that the match results in additional goals, the following lines pertain to the solution of these goals. If no match is possible or if the match results directly in a success, the line is further annotated with the word $FAIL$ or $SUCC$. In the former case, the succeeding lines indicate the solution attempt after backtracking and in the latter case they indicate an attempt to solve the remaining goals.

```
module m1.                  module m2.            module m3.
import m2.                  import m3.            type r i -> o.
(p X) :- (q X),(t X).       (p b).                (r a).
(t b).                      (q X) :- (s X).       (r b).
(s X) :- (r X).
```

Fig. 2. A Set of Interacting Modules

```
            m1 ?- (pX)
        m2, m1 ?- (q X)
    m3, m2, m1 ?- (s X)
m2, m3, m2, m1 ?- (r X)   X <- a   SUCC
        m2, m1 ?- (t a)   FAIL
m2, m3, m2, m1 ?- (r X)   X <- b   SUCC
        m2, m1 ?- (t b)   SUCC
```

Fig. 3. Solving ($m1$ ==> (p X)) Given the Modules in Figure 2

Let us consider now the attempt to solve the mentioned goal, ($m2$ ==> ((p X)). The initial program context is empty, but dealing with the module implication causes $m1$ to be added to it. The goal to be solved now is (p X). There is only one clause available for p and this is interpreted as if it were

(p X) :- ($m2$ ==> ((q X), (t X)))

since $m1$ imports $m2$. Module $m2$ is therefore added to the program context and the goal to be solved reduces to (q X), (t X). Although not relevant to the solution of the present goal, notice that module $m2$ also contains a clause for p. The new program context thus contains an enhanced definition for this predicate and an implementation must be capable of combining code from different sources to produce the desired effect. Tracing through the solution attempt a few more steps, we see that the use of the second clause in module $m1$ results in an attempt to solve (r X). There are two clauses for this predicate in the relevant program context and these are used in order. Note that this interaction could not have been predicted from the static structure of $m1$ alone: there is no compile-time indication that code in module $m3$ might be used in solving goals appearing in the bodies of clauses in $m1$. A compilation scheme must therefore be sensitive to the fact that the definition of procedures used within modules are determined dynamically. Continuing with the solution attempt, (r X) is solved successfully with X being bound to the constant a. The task now becomes one of solving the goal (t a). Notice that the program context for this goal includes only $m1$ and $m2$, *i.e.*, an implementation must support this kind of context switching. When this goal fails, backtracking now requires an alternative solution to (r X) to be found. However, this solution attempt must take place in a resurrected context, as indicated in the figure. Once again, an implementation of the module feature must be capable of supporting this kind of reinstatement of earlier contexts.

We consider in the next section the various implementation issues pertaining to the dynamic behavior of modules that are raised by the above example. We note that a desirable feature of an implementation scheme is that it should permit a separate compilation of each module; this is in some sense indicative of the ability of this feature to split up a program into logically separate parts. The scheme that we present for implementing the dynamic behavior exhibits this facet — separate segments of compiled code are produced for each module and these are linked together dynamically to produce a desired program context. However, the idea of separate compilation is somewhat more problematic at the level of static interaction. The main issue is that the parsing of an importing module requires the various type, kind and operator declarations in the imported modules, implying a dependence in compilation. This kind of behavior is, however, not unique to our context. The usual solution to this problem is to introduce the idea of an interface between modules. Specialized to our context, this involves assigning a set of declarations to a module name. This assignment may act in a prescriptive fashion on the actual set of declarations appearing in the module in the sense that they may be required to conform to the "interface" requirements. With regard to importation, on the other hand, the interface declarations could control what is visible. One consequence of this view is that the association of types with constants might be hidden. Such an occlusion must be accompanied with a hiding of the constant itself and thus affects the dynamic behavior. However, this behavior can be modelled by the use of implicit local declarations[3]. A proper use of this idea will require predicate definitions also to be hidden. This ability is not supported within the current language: the ability to quantify existentially over predicate names requires an extension of the syntax of $\mathcal{D}$-formulas. The extension in syntax can be easily accomplished as indicated in [4] and [10]. Although we do not treat this matter explicitly here, the desired extension does not cause any semantical problems and, as indicated in [16], can also be accommodated within our implementation scheme.

While the use of an interface as a method for prescribing interactions in this manner has several interesting aspects, a more conservative view of it is also possible. The interface declarations may be viewed simply as a distillate of the compilation of the module in question. Regardless of which view is taken, we assume here that, when a module is being compiled, all the type, kind and operator declarations obtainable from the imported modules are known. The scheme that we present in the next section then generates the code for capturing the dynamic behavior of a module by using only these interfaces and parsing the module in question. In this sense, our scheme is capable of supporting the idea of separate compilation.

4 Implementing the Dynamic Semantics of Modules

The crucial issue that must be dealt with in an implementation of the dynamic aspects of modules is the treatment of module implication. In particular, we are interested in the compilation of goals of the form M ==> G. Within a model that supports separate compilation, the production of code from the predicate definitions appearing in M must be performed independently of this goal. The compiled effect of this goal must then be to enhance the program context by

[3] A related proposal is contained in [12]. However, the suggestion there is to determine the local declarations dynamically, depending on the goal to be solved. This appears not to help with the "static" problem discussed here and also makes it difficult to generate code for a module independent of its use.

adding the code in M to it. Under this view, the symbol ==> becomes a primitive for linking code. The crucial issues within an implementation thus become those of what structures are needed for realizing this linking function and of what must be produced as a result of the compilation of a module to facilitate the linking process at run-time.

We have developed a scheme elsewhere [16] for implementing goals that contain implications. The dynamic semantics of module implication coupled with some features of the mentioned scheme make it an apt one to adapt to the present context. The essence of our scheme is to view a program as a composite of compiled code and a layered access function to this code. The execution of an implication goal causes a new layer to be added to an existing access function. Thus, consider an implication goal of the form $(C_1, \ldots, C_n)$ => G where, for $i \leq i \leq n$, C_i is a closed program clause of the form $\forall x_1 \ldots \forall x_n A_r$ or $\forall x_1 \ldots \forall x_n (G \supset A_r)$[4]. Each C_i corresponds to a partial definition of a procedure that must be added to the front of the program while an attempt is made to solve G. These clauses can be treated as an independent program segment and compiled in a manner similar to that employed in the WAM. Let us suppose that the clauses define the predicates $p_1, \ldots, p_r$. The compilation process then results in a segment of code with r entry points, each indexed with the name of a predicate. In our context, compilation must also produce a procedure that we call *find_code* that performs the following function: given a predicate name, this procedure returns the appropriate entry point in the code segment if the name is one of $p_1, \ldots, p_r$ and an indication of failure otherwise. This function can be implemented in several different ways such as through the use of a hash-function, but the details will not concern us here. Returning now to the implication goal, its execution results in a new access function that behaves as follows. Given a predicate name, *find_code* is invoked with it. If this function succeeds, then the code location that it produces is the desired result. Otherwise the code location is determined by using the access function in existence earlier.

The process of enhancing a context described above is incomplete in one respect: the new clauses provided for $p_1, \ldots, p_r$ may in fact be adding to earlier existing definitions for these predicates. To deal with this situation, the compilation process must produce code for each of these predicates that does not fail eventually, but instead looks for code for the relevant predicate using the access function existing earlier. Rather than carrying out this task each time it is needed, using an idea from [8], it can be done once at the time the new program context is set up. The idea used is the following. A vector of size r can be asso-

[4] In the general case, every implication goal can be transformed into one of the form

$$Q_1 X_1 \ldots Q_m X_m((C_1(X_1, \ldots, X_m), \ldots, C_n(X_1, \ldots, X_m)) \mathord{=}\!\!> G(X_1, \ldots, X_m))$$

where Q_i is $\exists$ or $\forall$ and $C_i(x_1, \ldots, x_m)$ is a program clause of the sort indicated but which may depend on the variables $x_1, \ldots, x_m$. Existential quantifiers may arise in considerations of module implication only if the module notion is enriched to allow for parameterization. Universal quantifiers do arise indirectly through *local* declarations whose treatment is considered later in this section.

ciated with the implication goal, with the ith entry in this vector corresponding to the predicate p_i. Now, the compilation of the body of the implication goal creates a procedure called *link_code* whose purpose is to fill in this vector when the implication goal is executed. This procedure essentially uses the name of each of the predicates and the earlier existing access function to compute an entry point to available code or, in case the predicate is previously undefined, to return the address of a failing procedure. To complement the creation of this table, the last instruction in the code generated for each of the predicates p_i must actually result in a transfer to the location indicated by the appropriate table entry.

In the framework of a WAM-like implementation, the layered access function described above can be realized by using what are called ***implication point records***. These records are allocated on the local stack and correspond essentially to layers in the access function. The components of such a record, based on the discussions thus far, are the following:

(i) the address of the *find_code* procedure corresponding to the antecedent of the implication goal,
(ii) a positive integer r indicating the number of predicates defined by the program clauses in the antecedent,
(iii) a pointer to an enclosing implication point record, and thereby to the previous layer in the access function, and
(iv) a vector of size r that indicates the next clause to try for each of the predicates defined in the antecedent of the implication goal.

The program context existing at a particular stage is indicated by a pointer to a relevant implication point record which is contained in a register called `I`. Now a goal such as $(C_1, \ldots, C_n)$ `=>` G is compiled into code of the form

```
push_impl_point t
        { Compiled code for G }
pop_impl_point
```

In this code, t is the address of a statically created table for the antecedent of the goal that indicates the address of its *find_code* and *link_code* procedures and the number of predicates defined. The *push_impl_point* instruction causes a new implication point record to be allocated. The first three components of this record are set in a straightforward manner using the table indicated and the contents of the `I` register. Filling in the last component involves running *link_code* using the access function provided by the `I` register. The final action of the instruction is to set the `I` register to point to the newly created implication point record. The effect of the *pop_impl_point* instruction is to reset the program context. This is achieved simply by setting the `I` register to the address of the enclosing implication point record, a value stored in the record the `I` register currently points to.

There are a few points about the scheme described that are worth mentioning. First, under this scheme the compilation of an atomic goal does not yield an instruction to transfer control to a particular code address. Rather, the instruction produced must use an existing access function (indicated by the I register) and an index generated from the name of the predicate to locate the relevant code. Notice that this behavior is to be anticipated, given the dynamic nature of procedure definitions. The second observation pertains to the resurrection of a context upon backtracking. Under our scheme, the program context is reduced to the contents of a single register. By saving these contents in a WAM-like choice point record and by retaining implication point records embedded within choice points, the necessary context switching can be easily achieved.

We turn finally to the implementation of module implication. Let us consider first the treatment of a module implication of the form M ==> G where M is a module with no local declarations and no import statements. From the perspective of dynamic semantics, M can be reduced to a conjunction of closed D-formulas of the form $\forall x_1 \ldots \forall x_n A_r$ or $\forall x_1 \ldots \forall x_n (G \supset A_r)$, *i.e.*, of the form just considered. Thus the scheme outlined above can be applied almost without change to the treatment of this kind of module implication. Under this scheme, the compilation of the module M must produce code that implements the relevant *find_code* and *link_code* procedures in addition to the compiled code for the various predicates defined. The linking operation corresponding to ==> effectively amounts to setting up an implication point record. The main task involved in this regard is that of executing the *link_code* function which in a sense links the predicate definitions in the module to those already existing in the program.

The handling of local declarations does not pose any major complications. The treatment of a goal of the form E => G that is indicated by the operational semantics essentially requires the existential quantifiers at the front of E to be replaced by new constants and the resulting D-formula to be added to the existing program. Implementing this idea results in the local constants in E being conceived of as constants but with a numeric tag that prevents them from appearing in terms substituted for logic variables in G. At a level of detail, these constants can be identified with cells in an implication point record and the *push_impl_point* instruction has the additional task of allocating these cells and of tagging them with the appropriate numeric value.

The only remaining issue is the treatment of import declarations. Let us assume that a module M imports the modules $M1$,$M2$ and $M3$. From the perspective of dynamic semantics, this importation has an effect largely on the clauses appearing in M. Let P :- G be one of these clauses. Based on the semantics of importing, this clause is to be interpreted as the clause

P :- ($M1$ ==> ($M2$ ==> ($M3$ ==> G))).

This translation actually indicates a straightforward method for implementing the effect of importation: the body of the clause can be compiled into the code generated for G nested within a sequence of *push_impl_point* and *pop_impl_point* instructions. Noting that module M may contain several clauses, an improvement is possible in this basic scheme. We identify with a module two additional functions that we call *load_imports* and *unload_imports*. In the case of module M, executing the first of these corresponds conceptually to executing the sequence

> *push_impl_point* $M1$
> *push_impl_point* $M2$
> *push_impl_point* $M3$

and, similarly, executing the second corresponds to executing a sequence of three *pop_impl_point* instructions. The address of these two functions is included in the implication point record created when a module is added to the program context. From the perspective of compilation, the code that is generated for the clause considered now takes the following shape:

```
{Code for unifying the head of the clause }
push_import_point M
        {Compiled code for goal G}
pop_import_point M
```

The *push_import_point* instruction in this sequence has the effect of invoking the *load_imports* function corresponding to module M and the *pop_import_point* instruction similarly invokes the *unload_imports* function.

The scheme described above assumes that the address of the compiled code and the various functions associated with a module can be indexed by the name of the module. This information is organized into entries in a global table with each entry having the following components:

(i) r, the number of predicates defined in the module,
(ii) the starting address for the compiled code segment for the predicates defined in the module; *find_code* will return offsets from this address,
(iii) the address of the *find_code* routine for the module,
(iv) the address of the *link_code* routine for the module,
(v) the address of the *load_imports* routine for the module, and
(vi) the address of the *unload_imports* routine for the module.

In reality not every module is loaded into memory at the beginning of a program and hence not every module has an entry in the global table. If a module that does not already reside in memory is needed, then a loading process brings the various segments of code in and creates an appropriate entry in the global table for the module. It should be clear by this point that the codes and information needed for each module can be obtained by a compile-time analysis of that module and the necessary interface definitions.

5 Controlling Redundancy in Search

The semantics presented for module implication and for the *import* statement could result in the same module being added several times to a program context. This has a potential drawback: it may result in redundancy in the search for a solution to a goal and the same solution may also be produced several times. To understanding this possibility, let us consider the following definition of a module called *sets* which imports the module *lists* presented in Section 3.

```
module sets.
import lists.
type subset (list A) → (list A) → o.
subset nil L.
(subset X :: L1 L2) :- ((member X L2), (subset L1 L2)).
```

Assume now that an attempt is made to solve the goal

$$sets \Longrightarrow subset\ 1 :: 2 :: 4 :: nil\ 1 :: 2 :: 3 :: nil.$$

Using the linear format described in Section 3, part of the effort in solving this goal is represented by the following sequence:

$$\begin{aligned}
sets\ ?\text{-}\ &(subset\ 1 :: 2 :: 4 :: nil\ 1 :: 2 :: 3 :: nil)\\
lists, sets\ ?\text{-}\ &(member\ 1\ 1 :: 2 :: 3 :: nil)\quad SUCC\\
lists, sets\ ?\text{-}\ &(subset\ 2 :: 4 :: nil\ 1 :: 2 :: 3 :: nil)\\
lists, lists, sets\ ?\text{-}\ &(member\ 2\ 1 :: 2 :: 3 :: nil)\\
lists, lists, sets\ ?\text{-}\ &(member\ 2\ 2 :: 3 :: nil)\quad SUCC\\
lists, lists, sets\ ?\text{-}\ &(subset\ 4 :: nil\ 1 :: 2 :: 3 :: nil)\\
lists, lists, lists, sets\ ?\text{-}\ &(member\ 4\ 1 :: 2 :: 3 :: nil)
\end{aligned}$$

It is easily seen that the attempt to solve the last goal in this sequence in the indicated program context will fail. Notice however, that a considerable amount of redundant search will be performed before this decision is reached: there are three copies of the module *lists* in the program context and the clauses for *member* in each of these will be used in turn in the solution attempt. A similar redundancy is manifest in the answers that are computed under the semantics provided. For instance, the query

$$sets \Longrightarrow subset\ S\ 1 :: 2 :: nil$$

will result in the substitution $1 :: 2 :: nil$ for S being generated twice through the use of the clauses in two different copies of the module *lists*.

The extra copies of the module *lists*, while leading to redundancy in search, do not result in an ability to derive new goals or to find additional answers. Adding these copies also results in a runtime overhead: given the implementation scheme of the previous section, the addition of each copy results in the creation of an implication point record, thereby consuming both space and time. A pragmatic question to ask, therefore, is whether the number of copies of any module in a program context can be restricted to just one. In answering this question there is an important principle to adhere to. It is desirable that the *logical semantics* of our language not be altered. In particular, we still want to be able to understand our language by using the derivation rules presented in Section 2 and to understand the dynamic semantics of modules through the devices discussed in Section 3. This principle is important because, as argued in [12], several interesting tools for analyzing the behavior of programs depend on this kind of a logical understanding of programming language constructs. In light of this principle, the question raised can be changed to one of the following sort: is it possible to preserve the important *observable* aspects of the given semantics while perhaps changing the details of the operational semantics so as to produce a preferred computational behavior. An affirmative answer to this question permits us to have the best of both worlds. The original semantics can be used for analyzing the interesting aspects of the behavior of programs while an actual implementation can be based on a modified set of derivation rules.

In the context being considered, the important aspects of program behavior are the set of queries that can be solved and the answers that can be found to any given query. Both aspects are completely determined by the set of sequents that have derivations. Thus, based on the above discussion, we might contemplate changing the underlying derivation rules for our language so as to reduce the *number* of derivations for any sequent while preserving the set of sequents that have derivations. With this in mind, we observe that the main source of redundancy in the example considered above is the *AUGMENT* rule. Assume that we want to solve a goal of the form D => G. The *AUGMENT* rule requires D to be added to the program context before attempting to solve G. Notice, however, that if D is already available in the program context, this addition is not likely to make a derivation of G possible where it earlier was not. A more interesting case is when the implication goal is of the form $(\exists x_1 \ldots \exists x_n D)$ => G. In this case the AUGMENT rule requires the addition of $D[c_1/x_1, \ldots, c_n/x_n]$ (for a suitable choice of c_is) to the program prior to the attempt to solve G. However, if the program already contains a clause of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$, the addition is again redundant from the perspective of being able to solve G.

In the rest of the section we prove the observations contained in the previous paragraph. Towards this end, we define an alternative to the AUGMENT rule.

Definition 5. Let G be a type variable free query and let $\mathcal{P}$ be a program. Then the AUGMENT′ rule is applicable if G is of the form $(\exists x_1 \ldots \exists x_n D)$ => G' and can be used to construct a derivation for $\mathcal{P} \longrightarrow G$ as follows:

(i) If a formula of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$ does not appear in $\mathcal{P}$, then by constructing a derivation for $D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P} \longrightarrow G'$ where, for $1 \leq i \leq n$, c_i is a nonlogical constant of the same type as x_i not appearing in the formulas in $\mathcal{P}, G$.

(ii) If a formula of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$ appears in $\mathcal{P}$, then by constructing a derivation for $\mathcal{P} \longrightarrow G'$.

Let us refer to the derivation rules presented earlier as $DS1$ and let $DS2$ be obtained from $DS1$ by replacing AUGMENT with AUGMENT′. We say that a sequent has a derivation in $DS1$ ($DS2$) if a derivation can be constructed for it by using the rules in $DS1$ (respectively, $DS2$). We now make the following observation about derivations in $DS2$.

Lemma 6. *Let G be a type variable free query, let D be a program clause whose free variables are included in $x_1, \ldots, x_n$ and let $\mathcal{P}_1$ and $\mathcal{P}_2$ be programs that between them contain a formula of the form $D[c'_1/x_1, \ldots, c'_n/x_n]$ where, for $1 \leq i \leq n$, c'_i is a nonlogical constant of the same type as x_i. Further, for $1 \leq i \leq n$, let c_i be a nonlogical constant of the same type as c'_i that do not appear in D. Finally, let $\mathcal{P}'_1$, $\mathcal{P}'_2$ and G' be obtained from $\mathcal{P}_1$, $\mathcal{P}_2$ and G, respectively, by replacing, for $1 \leq i \leq n$, c_i with c'_i. Now, if $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G$ has a derivation of length l in DS2, then there must also be a derivation in DS2 for $\mathcal{P}'_1, \mathcal{P}'_2 \longrightarrow G'$ that is of length l or less.*

Proof. We prove the lemma by an induction on the length of the derivation in DS2 of the first sequent. If this derivation is of length 1, it must have been obtained by using the SUCCESS rule. Now, if G is equal to an instance of $D[c_1/x_1, \ldots, c_n/x_n]$, then G' must be equal to an instance of $D[c'_1/x_1, \ldots, c'_n/x_n]$. Further, if G is an instance of a clause in $\mathcal{P}_1$ or in $\mathcal{P}_2$ it must be the case that G' is an instance of a clause in $\mathcal{P}'_1$ or in $\mathcal{P}'_2$. From these observations it follows that the SUCCESS rule is applicable to $\mathcal{P}'_1, \mathcal{P}'_2 \longrightarrow G'$ as well and so this sequent also must have a derivation of length 1.

Suppose now that the derivation of $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G$ is of length $(l+1)$. We assume that the requirements of the lemma are satisfied by all sequents that have derivations of length l or less and show this must also be the case for the sequent being considered. The argument proceeds by examining the possible cases for the first rule used in the derivation in question.

Let us assume that this rule is an *AND*. In this case G must be of the form $G_1 \wedge G_2$ and there must be derivations of length l or less for the sequents $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1$ and $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_2$. By hypothesis, there are derivations of length l or less for $\mathcal{P}'_1, \mathcal{P}'_2 \longrightarrow G_1$ and $\mathcal{P}'_1, \mathcal{P}'_2 \longrightarrow G_2$. Using these derivations together with an *AND* rule, we obtain one of length $l+1$ or less for $\mathcal{P}'_1, \mathcal{P}'_2 \longrightarrow G'_1 \wedge G'_2$. Now, G' must be equal to the formula $G'_1 \wedge G'_2$. Thus the desired conclusion is obtained in this case.

Arguments similar to that for *AND* can be supplied for the cases when OR or INSTANCE is the first rule used. In the case that GENERIC is used, G must be of the form $\forall y G_1$ and there must be a derivation of length l for

$$\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1[a/y]$$

for some nonlogical constant a of the same type as y that does not appear in G, $D[c_1/x_1, \ldots, c_n/x_n]$ or in the formulas in $\mathcal{P}_1$ and $\mathcal{P}_2$. We can almost use an argument similar to that employed for AND. The only problem is that a might be identical to some c'_i for $1 \leq i \leq n$. However, the following fact is easily seen: a derivation of length l for a sequent Ξ can be transformed into one of identical length for a sequent obtained from Ξ by replacing all occurrences of a nonlogical constant b with some other (nonlogical) constant of the same type. Using this together with the "newness" condition on a, we may assume that a is distinct from all the c'_is. The argument in this case can then be completed without trouble.

In the case that the first rule employed is BACKCHAIN, a combination of the observations used for SUCCESS and AND must be employed. In particular, let G_1' be the result of replacing, for $1 \leq i \leq n$, all occurrences of c_i by c_i' in G_1. Now, if $G_1 \supset G$ is an instance of $D[c_1/x_1, \ldots, c_n/x_n]$, then $G_1' \supset G'$ must be an instance of $D[c_1'/x_1, \ldots, c_n'/x_n]$. Further, if $G_1 \supset G$ is an instance of a program clause in $\mathcal{P}$, then $G_1' \supset G'$ must also be an instance of the same clause. Finally, using the hypothesis, if $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1$ has a derivation of length l, then $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_1'$ has a derivation of length l or less. Using these various facts, it is easily seen that if the first rule used in the derivation for $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G$ is BACKCHAIN, then a derivation can be provided for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow G'$ in which the last rule is once again a BACKCHAIN and, further, this derivation will satisfy the length requirements.

Suppose now that the first rule used is AUGMENT$'$ and that case (i) of this rule is the applicable one. Then G must be of the form $(\exists y_1 \ldots \exists y_m D_1) \supset G_1$ and further, no formula of the form $D_1[a_1'/y_1, \ldots, a_m'/y_m]$ must appear in $\mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2$. By assumption, there is a derivation of length l for

$$D_1[a_1/y_1, \ldots, a_m/y_m], \mathcal{P}_1, D[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}_2 \longrightarrow G_1$$

where, for $1 \leq i \leq m$, a_i is a constant of appropriate type and meeting the needed requirements of newness. By an argument similar to that used in the case of BACKCHAIN, we can assume that the a_js are distinct from the c_is and the c_i's. Then, using the induction hypothesis, there must be a derivation of length l or less for

$$D_1'[a_1/y_1, \ldots, a_m/y_m], \mathcal{P}_1', \mathcal{P}_2' \longrightarrow G_1'$$

where D_1' and G_1' are obtained from D_1 and G_1 by the replacement, for $1 \leq i \leq n$, of c_i by c_i'. Now, if a formula of the form $D_1[a_1'/y_1, \ldots, a_m'/y_m]$ did not appear in $\mathcal{P}_1$ or $\mathcal{P}_2$, then one of the form $D_1'[a_1'/y_1, \ldots, a_m'/y_m]$ cannot appear in $\mathcal{P}_1'$ or $\mathcal{P}_2'$. Thus, the derivation of the indicated sequent can be used together with an AUGMENT$'$ rule to obtain one for $\mathcal{P}_1', \mathcal{P}_2' \longrightarrow (\exists y_1 \ldots \exists y_m D_1') \supset G_1'$; a newness condition has to be satisfied by $a_1, \ldots, a_m$ for the AUGMENT rule to be used, but this can be seen to be the case, using particularly the assumption of distinctness from the c_i's. The derivation of the last sequent is obviously of length $(l+1)$ or less. Observing that $(\exists y_1 \ldots \exists y_m D_1') \supset G_1'$ is the same formula as G', the lemma is seen to hold in this case.

The only situation remaining to be considered is that when the first rule corresponds to case (ii) of AUGMENT$'$. The argument in this case is similar to that employed for case (i) of the same rule. The details are left to the reader.

Using the above lemma we now show the equivalence of $DS1$ and $DS2$ from the perspective of derivability of sequents of the kind we are interested in.

Theorem 7. *Let $\mathcal{P}$ be a program and let G be a type variable free query. There is a derivation for $\mathcal{P} \longrightarrow G$ in $DS1$ if and only if there is a derivation for the same sequent in $DS2$.*

Proof. Consider first the forward direction of the theorem. The only reason why the derivation in $DS1$ might not already be one in $DS2$ is because the AUGMENT rule that is used is in some cases not an instance of the AUGMENT′ rule. Consider the last occurrence of such a rule in the derivation. In this case, a derivation is constructed for a sequent of the form $\mathcal{P}' \longrightarrow (\exists x_1 \ldots \exists x_n D') \supset G'$ from one for the sequent $D'[c_1/x_1, \ldots, c_n/x_n], \mathcal{P}' \longrightarrow G'$, where the c_is are appropriately chosen constants. Given that we are considering the last occurrence of an errant rule, the derivation for the latter sequent must be one in $DS2$ as well. Since the application of the AUGMENT rule being considered does not conform to the requirements of the AUGMENT′ rule, it must be the case that, for some choice of constants $c'_1, \ldots, c'_n$, $D'[c'_1/x_1, \ldots, c'_n/x_n]$ appears in $\mathcal{P}'$. But then, using Lemma 6 and the fact that the constants $c_1, \ldots, c_n$ must not appear in G' or in the formulas in $\mathcal{P}'$, we see that $\mathcal{P}' \longrightarrow G'$ has a derivation in $DS2$. Using this derivation together with case (i) of the AUGMENT′ rule, we obtain a derivation in $DS2$ for the original sequent, *i.e.*, for $\mathcal{P}' \longrightarrow (\exists x_1 \ldots \exists x_n D') \supset G'$. We repeat this form of argument to ultimately transform the derivation in $DS1$ for $\mathcal{P} \longrightarrow G$ into one in $DS2$.

To show the theorem in the reverse direction, we observe the following fact: for any program $\mathcal{P}'$, type variable free query G' and program clause D', if $\mathcal{P}' \longrightarrow G'$ has a derivation in $DS1$, then $D', \mathcal{P}' \longrightarrow G'$ also has a derivation in $DS1$. Now, a derivation in $DS2$ may not be a derivation in $DS1$ only because case (i) of AUGMENT′ was used in some places. However, this can be corrected by using the observation just made. In particular, we consider the last occurrence of an errant rule in the derivation and convert it into an occurrence of the AUGMENT rule by using the above fact. A repeated use of this transformation yields the theorem.

An easy consequence of the above theorem is the following:

Corollary 8. *Let $\mathcal{P}$ be a program and let G be a query. The set of answers to G in the context of $\mathcal{P}$ is independent of whether rules in $DS1$ or in $DS2$ are used in constructing derivations.*

We have thus shown that, from the perspective of solving queries and computing answers, it is immaterial whether the rules in $DS1$ or those in $DS2$ are used to construct derivations. By virtue of Proposition 4, we can in fact use the notion of intuitionistic derivability for the purpose of analyzing programs in our language while using the rules in $DS2$ to carry out computations. At a pragmatic level, there is a definite benefit to using the AUGMENT′ rule instead of the AUGMENT rule in solving queries, since considerable redundancy in search can be eliminated by this choice. We use this observation to yield a more viable implementation of module implication and of the *import* statement in the next section. We note that another approach to controlling the redundancy arising out of the module semantics is suggested in [12]. However, this approach is less general than the one considered here in that it applies only to *import* statements and not to module implications. Moreover, the correctness of the approach is only conjectured in [12]. The observations in this section can be used in a straightforward fashion to verify this conjecture.

6 An Improved Implementation of Modules

We now consider an implementation of our language that uses the AUGMENT′ rule instead of the AUGMENT rule whenever possible. Under the new rule, solving an goal of the form $(\exists x_1 \ldots \exists x_n D) \supset G$ requires checking if there is already a clause of the form $D[c_1/x_1, \ldots, c_n/x_n]$ in the program. Clearly an efficient procedure for performing this test is a key factor in using the changed rule in an actual implementation. It is difficult to achieve this goal in general. One problematic case is when the goal $(\exists x_1 \ldots \exists x_n D) \supset G$ arises as part of a larger goal and D contains variables that are bound only in this larger context. Instantiations for these variables may be determined in the course of execution, thus making it difficult to perform the desired test by a simple runtime operation. In fact, the device of delaying instantiations might even make it impossible to determine the outcome of the test at the time the implication goal is to be solved because "clauses" in the program might contain logic variables. An example of this kind was seen in Section 2. The attempt to solve the goal $\exists L(rev\ 1 :: 2 :: nil\ L)$ resulted there in the clause $(rev\_aux\ nil\ L)$ being added to the program. The precise shape of this clause clearly depends on the instantiation chosen for L. A test of the sort needed by AUGMENT′ cannot be performed with regard to this clause prior to this shape being determined.

The above discussion demonstrates that the optimization embodied in the AUGMENT′ rule can be feasibly implemented only relative to a restricted class of program clauses, namely, clauses that do not contain logic variables. Of particular interest from this perspective is a statically identifiable closed E-formula that has the potential for appearing repeatedly in the antecedent of implication goals. Given such a formula E, a mark can be associated with it that records whether or not the current goal is dynamically embedded within the invocation of an implication goal of the form E => G. If it is so embedded and if the current goal is itself of the form E => G', then, in accordance with the AUGMENT′ rule, the computation can proceed directly to solving G' without affecting additions to the program.

The dynamic semantics of module implication provides a particular case of the kind of formula discussed above, namely the (closed) E-formula identified with a module. Thus, assume that we are trying to solve the goal M ==> G. If we know that the module M has already appeared in the antecedent of a module implication goal within which the current one is dynamically embedded, then no enhancements to the program need be made. The implementation scheme presented in 4 provides a setting for incorporating this test in an efficient manner. The essential idea is that we include an extra field called *added* in the record in the global table corresponding to each module. This field determines whether or not the clauses in a particular module have been added to the program in the path leading up to the current point in computation. When the goal M ==> G is to be solved, the *added* field for M's entry in the global table is checked. If this indicates that the clauses in M has not previously been added, then the addition is performed and the status of the field is changed. Otherwise the computation proceeds directly to solving G.

While the idea described above is simple, some details have to be paid attention to in its actual implementation. One issue is the action to be taken on the completion of a module implication goal. At a conceptual level, the successful solution of the goal M ==> G must be accompanied by a removal of the code for M; this is accomplished in our earlier scheme by the instruction *pop_impl_point*. However, given the current approach, an actual removal must complement only an actual addition. To facilitate a determination of the right action to be taken, the *added* field is implemented as a counter rather than as a boolean. This field is initialized to 0. Each time a module is conceptually added to the program context, its *added* value is incremented. A conceptual removal similarly causes this value to be decremented. An actual removal is performed only when the counter value reaches 0.

The second issue that must be considered is the effect of backtracking. As we have noted, this operation might require a return to a different program context. An important characteristic of a program context now is the status of the *added* fields, and backtracking must set these back to values that existed at an earlier computation point. To permit an accomplishment of this resetting action, changes made to this field must be trailed. A naive implementation would trail the old value every time a change needs to be made, *i.e.*, every time a module is added or removed. A considerable improvement on this can be obtained by trailing a value only if there is a possibility to return to a state in which it is operative. Thus consider a goal of the form

m ==> $(G1, (m$ ==> $G2))$

When the *added* field for m is incremented for the second time, there is a need to trail the old value only if unexplored alternatives exist in the attempt to solve $G1$. There is a simple way to determine this within a WAM-like implementation. Let us suppose we record the address of the most recent choice point at the time of processing the outermost (module) implication in the global table entry corresponding to m. Now, when the embedded implication is processed, we compare the address of the current most recent choice point with the recorded value. There is a backtracking point in the solution of $G1$ only if the first is greater than the second. Similarly, consider the decrement that is made to the *added* field when a goal of the form m ==> G is completed. The old value needs to be trailed only if choice points exist within the solution for G. A test identical to that described above suffices to determine whether this is the case.

In order to implement the above idea, one more field must be added to the entries in the global table for modules, *i.e.*, one that records the most recent choice point prior to the latest change to the *added* field. This field is called *mrcp* and is initialized to the bottom of the stack. Notice that this field needs to be updated each time *added* has to be trailed, and this change must also be trailed. Accordingly, each cell in the trail introduced for managing the *added*

```
pushimpl(m)
begin
    if m.added = 0
    then create an implication point record for m;
    if m.mrcp < B
    then
    begin
        trail (m, m.mrcp, m.added);
        m.mrcp := B;
    end;
    m.added := m.added + 1
end;

popimpl(m)
begin
    if m.mrcp < B
    then
    begin
        trail (m, m.mrcp, m.added);
        m.mrcp := B;
    end;
    m.added := m.added - 1;
    if m.added = 0 then
        Set I to most recent implication point
        in record pointed to by I
end
```

Fig. 4. Adding and Removing Modules from Program Contexts

values contains three items: the name of a module, the old value of *added*, and the old contents of the *mrcp* field. Pointers to this trail must be maintained in choice points and the trail must be unwound in the usual fashion upon backtracking. Module implication is compiled as before, although the interpretation of *push_impl_point m* and *pop_impl_point m* changes. In particular, these can be understood as though they are invocations to the procedures *pushimpl*(m) and *popimpl*(m) that are presented in pseudo-code fashion in Figure 4. In this code we write *m.mrcp* and *m.added* to denote, respectively, the *mrcp* and *added* fields in the global table entry corresponding to the module m. We also recall that the B register in the WAM setting indicates the most recent choice point.

There is an auxiliary benefit to two fields that has been added under the present scheme to the records in the global table. As mentioned in Section 4, our implementation permits modules to be loaded on demand, and hence does not require all modules to be available in main memory during a computation. A question that arises is whether modules can also be unloaded to reclaim code space. This unloading must be done carefully because a module not currently included in the program context might still be required because of the possibility of backtracking. A quick check of whether a module can be unloaded is obtained by examining the two new fields in the global table entry for a module. If the *mrcp* field points to the bottom of the stack and *added* is 0, then the module is not needed and can be unloaded.

The implementation of the dynamic effects of *import* can, in principle, be left unchanged. However, a significant efficiency improvement can be obtained by noting the following: once a clause from a module *m* has been used by virtue of the BACKCHAIN rule, there is no further need to check if the modules imported by *m* have been added to the program context. To utilize this idea, we include two more fields in each implication point record:

(i) A field called *backchained* that records the number of times a clause from the module to which the implication point record corresponds has been backchained upon.
(ii) A field called *mrcp* that records the most recent choice point prior to the last change to *backchained*.

When the implication point record is created, the *backchained* field is initialized to 0 and the *mrcp* field is set to point to the bottom of the stack. Whenever a clause from a module corresponding to the implication point record is backchained upon, a conceptual addition of the imported modules must be performed. An actual addition must be contemplated within the present scheme only if the *backchained* field is 0. In any case, this field is incremented before the "body" of the clause is invoked. The increment to *backchained* is complemented by a decrement when the clause body has been successfully solved. Finally, an actual removal of the imported modules from the program context must be contemplated only when *backchained* becomes 0 again. For the purpose of backtracking, it may be necessary to trail an old value of *backchained* each time this field is updated. The *mrcp* field is useful for this purpose. Essentially, we compare this field with the address of the current most recent choice point, obtained in the WAM context from the B register. If the latter is greater than the *mrcp* field, then the old value of *backchained* must be trailed. This action must also be accompanied by a trailing of the existing *mrcp* value and the update of this field to the address of the current most recent choice point.

The rationale for the various actions described for handling imports is analogous to that in the case of module implication, and should be clear from the preceding discussions. At a level of detail, another trail is needed for maintaining the old values of the *backchained* and *mrcp* fields. The cells in this trail correspond once again to triples: the address of the relevant implication point record and the *backchained* and *mrcp* values. Pointers to this trail must also be maintained in choice points and backtracking must cause the trail to be unwound. The compilation of clauses in modules is performed as before: the code produced for the body of a clause in module *m* must be embedded within the instructions *push_import_point m* and *pop_import_point m*. These instructions can be understood as though they are invocations to the procedures *pushimport*(*m*) and

```
pushimport(m)
begin
    if CI.backchained = 0
    then call load_imports for m
    if CI.mrcp < B
    then
    begin
        trail ⟨CI,CI.mrcp,CI.backchained⟩;
        CI.mrcp := B;
    end;
    CI.backchained := CI.backchained + 1
end;

popimport(m)
begin
    if CI.mrcp < B
    then
    begin
        trail ⟨CI,CI.mrcp,CI.backchained⟩;
        CI.mrcp := B;
    end;
    CI.backchained := CI.backchained - 1;
    if CI.backchained = 0 then
        invoke unload_imports for m
end
```

Fig. 5. Adding Imported Modules to a Program Context

popimport(m) that are presented, in pseudo-code fashion, in Figure 5. Use is made in these procedures of a register called CI that points within our implementation to the implication point record from which the clause currently being considered is obtained. Further, we write CI.*mrcp* and CI.backchained to denote the *mrcp* and *backchained* fields in the implication point record that *CI* points to.

It is important to note that once a clause from a module has been backchained upon, the two instructions *push_import_point* and *pop_import_point* incur very little overhead with respect to clauses in that module. In particular, at most two tests, a trailing and two updates are necessary for each instruction. This is much less work than the creation of implication point records that was necessary under a direct implementation of the operational semantics. Further, this overhead appears to be acceptable even if these instructions are executed repeatedly.

We consider an example to illustrate the manner in which redundancy is controlled within the changed implementation. Let us assume that the modules $m0$, $m1$ and $m2$ are defined as below.

module $m0$.	*module* $m1$.	*module* $m2$.
import $m1, m2$.	*import* $m2$.	*kind i type*.
type $p\ i \rightarrow o$.	*type* $q\ i \rightarrow o$.	*type* $a\ i$.
$(p\ X)$:- $(q\ X), (t\ X)$.	$(q\ X)$:- $(r\ X)$.	*type* $b\ i$.
$(r\ X)$:- $(s\ X)$.		$(r\ a)$.
		$(s\ b)$.
		$(t\ b)$.

The attempt to solve the goal $m0$ ==> $(p\ X)$ is presented below. We augment the linear format of Section 3 as follows in this presentation: Each module in the program context is presented by a pair consisting of its name and the value of the *backchained* field in the implication point record created for it. At the end of each line, a list of pairs is presented that indicates module names and the values of the *added* field in the global table entry for each of them.

$(m0, 0)$?- $(p\ X)$		$[(m0, 1), (m1, 0), (m2, 0)]$
$(m2, 0), (m1, 0), (m0, 1)$?- $(q\ X)$		$[(m0, 1), (m1, 1), (m2, 1)]$
$(m2, 0), (m1, 1), (m0, 1)$?- $(r\ X)$ X <- a	*SUCC*	$[(m0, 1), (m1, 1), (m2, 2)]$
$(m2, 0), (m1, 0), (m0, 1)$?- $(t\ a)$ *FAIL*		$[(m0, 1), (m1, 1), (m2, 1)]$
$(m2, 0), (m1, 1), (m0, 1)$?- $(r\ X)$		$[(m0, 1), (m1, 1), (m2, 2)]$
$(m2, 0), (m1, 1), (m0, 2)$?- $(s\ X)$ X <- b	*SUCC*	$[(m0, 1), (m1, 1), (m2, 2)]$
$(m2, 0), (m1, 0), (m0, 1)$?- $(t\ b)$ *SUCC*		$[(m0, 1), (m1, 1), (m2, 1)]$

An interesting point to note in this computation is that the clause $(r\ a)$ in module $m2$ is used only once in solving the subgoal $(r\ X)$ even though there are conceptually two copies of $m2$ in the program context when the subgoal is invoked. Similarly, an attempt to find another solution to the query will fail, even though the same solution can be found five more times under a naive interpretation of the given semantics.

7 Conclusion

We have examined a notion of modules for the logic programming language λProlog in this paper. The notion considered provides a means for structuring the two components that determine programs in this language: the type, kind and operator declarations and the procedure definitions. Using a module typically involves making its contents available in some other context. As explained in some detail, this operation has static and dynamic effects within λProlog. Our focus here has been on the implementation of the dynamic aspects of modules. At a level of detail, we have proposed an implementation method that is based on a WAM-like machine and that has several interesting features:

(i) It supports the idea of compiling modules separately. In particular, the compilation of a module produces WAM-like code based on only the program clauses appearing in the module.

(ii) Interpreting a logical operation as a primitive for linking a module into a given program context, it uses a compilation process to generate linking code and includes run-time structures for accomplishing the linking function.

(iii) Based on a theoretical analysis of this notion, it includes mechanisms for reducing redundancy inherent in the given dynamic semantics of the module feature. The redundancy check is based on a two-level test that in the usual situation can be carried out with very little overhead.

There are several significant enrichments to a Prolog-like language that are embodied in λProlog in addition to the module feature. A complete implementation of this language must include mechanisms for dealing with all these features. As mentioned already, a detailed consideration has been given to the features other than the module notion elsewhere, resulting in an abstract machine for the core language described in Section 2. An actual implementation of this machine is currently being undertaken. The mentioned machine is entirely compatible with the ideas for handling modules that are presented in this paper and we plan to include these ideas within our implementation effort in the near future.

8 Acknowledgements

Work on this paper has been supported by NSF Grant CCR-89-05825.

References

1. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
2. Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.
3. Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.
4. Elsa L. Gunter. Extensions to logic programming motivated by the construction of a generic theorem prover. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 223–244. Springer-Verlag, 1991. Volume 475 of *Lecture Notes in Artificial Intelligence*.
5. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.
6. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
7. Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. Submitted, August 1992.
8. Evelina Lamma, Paola Mello, and Antonio Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In *Sixth International Logic Programming Conference*, pages 303–317, Lisbon, Portugal, June 1989. MIT Press.

9. Dale Miller and Gopalan Nadathur. λProlog version 2.7. Distributed in C-Prolog and Quintus Prolog source code, August 1988.
10. Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.
11. Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.
12. Dale Miller. A proposal for modules in λProlog. In *Workshop on the λProlog Programming Language*, Philadelphia, July 1992.
13. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
14. Luís Monteiro and António Porto. Contextual logic programming. In *Sixth International Logic Programming Conference*, pages 284–299, Lisbon, Portugal, June 1989. MIT Press.
15. Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.
16. Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Submitted, May 1992.
17. Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features. Submitted, November 1992.
18. Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. To appear in the *Journal of Automated Reasoning*.
19. Richard O'Keefe. Towards an algebra for constructing logic programs. In *1985 Symposium on Logic Programming*, pages 152–160, Boston, 1985.
20. D.T. Sannella and L.A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147 – 178, January 1992.
21. D.H.D. Warren. An abstract Prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.